

Self-healing using virtual structures

Amitabh Trehan *

Modern networks have evolved to become both large and highly complex, with some networks spanning nations and even the globe. Networks provide a multitude of services using a wide variety of protocols and components to the extent that they have now begun to resemble self-governed living entities. Most modern networks are dynamic with nodes entering the network or leaving by choice, failure or attack. There are dynamic networks which have always been around in some form, like social networks, which we have only now begun to analyze and in fact, influence. That maintaining robustness in modern networks can be an issue can be ascertained by the regular breakdowns in large and important networks e.g. the crash of the Skype network in 2007 [7, 10, 11, 13, 15] attributed to the failure of its “self-healing” mechanisms [1]. Also, due to the scale and nature of design of such networks, it may simply not be practical to build robustness into the individual nodes or into the structure of the initial network itself. Thus, the need for a responsive approach to robustness. Many important networks are also *reconfigurable* in the sense that they can change their topology e.g. peer-to-peer, wireless, ad-hoc networks and friendship networks on social networking sites etc. . We exploit this property of networks to allow us a responsive approach towards robustness. Moreover, our algorithms are scalable since our repair costs are constant or at most logarithmic in the number of nodes, and inherently handle the dynamism of the network. This chapter is organized as follows: Section 1 introduces self-healing and our model. Section 2 introduces virtual structures and discusses their use in self-healing. The subsequent sections(Sections 3 and 4) discuss in brief two of our algorithms which use virtual graphs: ForgivingTree [9] and ForgivingGraph [8], followed by a conclusion(Section 5).

1 What is self-healing?

Informally, self-healing is the maintenance of certain properties within desirable bounds by the nodes in a network suffering from failures or under attack. As the name implies, self-healing has to be initiated and executed by the nodes themselves. As such, the algorithms we have proposed here are fully distributed. Equivalently, we can say that a self-healing system, when starting from a correct state, can only be temporarily out of a correct state i.e. it recovers to a correct state, in presence of attacks. Self-healing is one of the so called ‘Self-*’ properties which systems such as autonomic systems may be required to have. In the distributed systems world, perhaps the most well-known self-* property is *self-stabilization* [4, 5, 6, 16]. Self-stabilization was introduced by Dijkstra in 1974 [4]. A self-stabilizing system is a system which, starting from an arbitrary state and being affected

*Information Systems Group, Faculty of Industrial Engineering and Management, Technion, Haifa, Isreal.
email: amitabh.trehaan@gmail.com

by adversarial transient failures, can, in finite time, recover to a correct state. Often, self-stabilization does not take code corruption (byzantine behavior) or fail-stop failures (node crashes) into account. A self-healing system, when starting from a correct state, can only be temporarily out of a correct state i.e. it recovers to a correct state, in presence of some adversarial attacks including node removal. Other self-* properties, often broadly defined, include *self-scaling*, *self-repairing* (similar to self-healing), *self-adjusting* (similar to self-managing), *self-aware/self-monitoring*, *self-immune*, *self-containing* [2].

Our sense of self-healing is more formally captured by the model discussed in Section 1.1. Informally, the model we adopt in this work is as follows. We assume that the network is initially a connected graph over n nodes. An adversary repeatedly attacks the network. This adversary knows the network topology and our algorithm, and it has the ability to delete arbitrary nodes from the network or insert a new node in the system (except for ForgivingTree(Section 3)) which it can connect to any subset of the nodes currently in the system. However, we assume the adversary is constrained in that in any time step it can only delete or insert a single node. Following that, the self-healing algorithm has a short time to reconfigure and heal the network by adding edges between remaining nodes before the next act of the adversary. Our model could, for example, capture what can happen when a worm or software error propagates through the population of nodes.

1.1 Model of self-healing

Our general model of self-healing is shown in Figure 1. This model was introduced in [17]. It is generalized from the model in [8]. Somewhat similar models were also used in [12, 9, 14]. The specific models used in most of our algorithms are special cases of this model, differing mainly in the way the success metrics of the graph properties are presented. The model used in Xheal [12] also differs in the synchronicity and message assumptions. Let $G = G_0$ be an arbitrary graph on n nodes, which represent processors in a distributed network. In each step, the adversary either deletes or adds a node. After each deletion, the algorithm gets to add some new edges to the graph, as well as deleting old ones. At each insertion, the processors follow a protocol to update their information. The algorithm's goal is to maintain the chosen graph properties within the desired bounds. At the same time, the algorithm wants to minimize the resources spent on this task. Initially, each processor only knows its neighbors in G_0 , and is unaware of the structure of the rest of G_0 . After each deletion or insertion, only the neighbors of the deleted or inserted vertex are informed that the deletion or insertion has occurred. After this, processors are allowed to communicate by sending a limited number of messages to their direct neighbors. We assume that these messages are always sent and received successfully. The processors may also request new edges be added to the graph. The only synchronicity assumption we make (in algorithms besides Xheal) is that no other vertex is deleted or inserted until the end of this round of computation and communication has concluded. To make this assumption more reasonable, the per-node communication cost should be very small in n (e.g. at most logarithmic).

We also allow a certain amount of pre-processing to be done before the first attack occurs. This may, for instance, be used by the processors to gather some topological information about G_0 , or perhaps to coordinate a strategy. Another success metric is the amount of computation and communication needed during this preprocessing round. For our success metrics, we compare the graphs at time T : the actual graph G_T to the graph G'_T which

is the graph with only the original nodes (those at G_0) and insertions without regard to deletions and healing. This is the graph which would have been present if the adversary was not doing any deletions and (thus) no self-healing algorithm was active. This is the natural graph for comparing results. Figure 2 shows an example of G'_T and a corresponding G_T . The figure also shows, in G'_T , the nodes and edges inserted and deleted, and in G_T , the edges inserted by the healing algorithm, as the network evolved over time. Table 1 compares our self-healing algorithms in this line of work: DASH [14], ForgivingTree [9], ForgivingGraph [8], Xheal [12].

Figure 1: The general distributed Node Insert, Delete and Network Repair Model.

<p>Each node of G_0 is a processor.</p> <p>Each processor starts with a list of its neighbors in G_0.</p> <p>Pre-processing: Processors may exchange messages with their neighbors.</p> <p>for $t := 1$ to T do</p> <p style="padding-left: 20px;">Adversary deletes a node v_t from G_{t-1} or inserts a node v_t into G_{t-1}, forming H_t.</p> <p style="padding-left: 20px;">if node v_t is inserted then</p> <p style="padding-left: 40px;">The new neighbors of v_t may update their information and exchange messages with their neighbors.</p> <p style="padding-left: 20px;">if node v_t is deleted then</p> <p style="padding-left: 40px;">All neighbors of v_t are informed of the deletion.</p> <p style="padding-left: 20px;">Recovery phase:</p> <p style="padding-left: 40px;">Nodes of H_t may communicate (asynchronously, in parallel) with their immediate neighbors. These messages are never lost or corrupted, and may contain the names of other vertices.</p> <p style="padding-left: 40px;">During this phase, each node may add edges joining it to any other nodes as desired.</p> <p style="padding-left: 40px;">Nodes may also drop edges from previous rounds if no longer required.</p> <p style="padding-left: 20px;">At the end of this phase, we call the graph G_t.</p>	<p>Success metrics: Minimize the following “complexity” measures:</p> <p>Consider the graph G' which is the graph consisting solely of the original nodes and insertions without regard to deletions and healings. Graph G'_t is G' at timestep t (i.e. after the t^{th} insertion or deletion).</p> <ol style="list-style-type: none"> 1. Graph properties/invariants. The graph properties/ invariants we are trying to preserve. e.g. <i>Degree increase</i>: $\max_{v \in G} \text{degree}(v, G_T) / \text{degree}(v, G'_T)$ 2. Communication per node. The maximum number of bits sent by a single node in a single recovery round. 3. Recovery time. The maximum total time for a recovery round, assuming it takes a message no more than 1 time unit to traverse any edge and we have unlimited local computational power at each node.
--	---

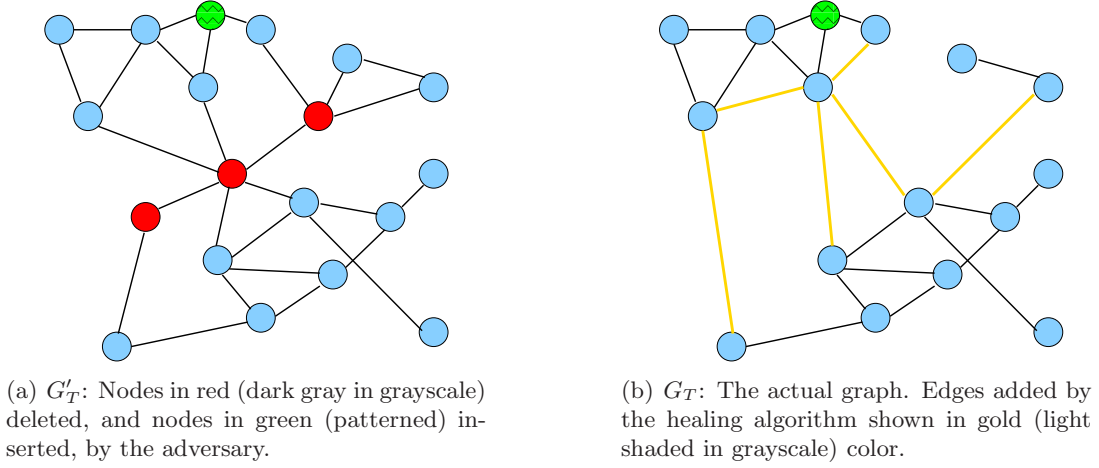


Figure 2: Graphs at time T . G'_T : The graph of initial nodes and insertions over time, G_T : The actual healed graph.

	Adversarial Attack		Property bounded			
	Deletion	Insertion	Connectivity	Degree (orig: d)*	Diameter (orig: D)*	Stretch
DASH	✓	✓	✓	$d + 2 \log n$	—	—
Forgiving Tree [†]	✓	×	✓	$d + 3$	$D \log \Delta$	—
Forgiving Graph [†]	✓	✓	✓	$3d$	$D \log n$	$\log n$
Xheal	✓	✓	✓	κd	$D \log n$	$\log n$

* ‘orig:’ the original value of the property in the graph (i.e. the value in the graph G' in our model)

	Costs				
	Repair time	# Msgs per deletion	Msg size	match lower bound [‡]	locality (hops) [#]
DASH	$O(\log n)$ [†]	$O(\delta \log n + \log^2 n)$ [†]	$O(\log n)$	✓	1
Forgiving Tree [†]	$O(1)$	$O(\delta)$	$O(\log n)$	✓	2
Forgiving Graph [†]	$O(\log \delta \log n)$	$O(\delta \log n)$	$O(\log^2 n)$	✓	$\log n$
Xheal	$O(\log n)$	$O(\kappa \log n A(p)/p)$ [§]	$O(\log^2 n)$	✓	$\log n$

[†] with high probability, and amortized over $O(n)$ deletions.

[‡] The lower bounds differ according to the properties being bounded.

[#] Number of hops from the deleted node to nodes involved in repair.

[§] Amortized over p deletions. $A(p)$ is a function of p , and k is a parameter.

[†] Algorithms using Virtual graphs

Table 1: Comparison of our self-healing Algorithms. d is the degree of an individual node, Δ is the maximum degree of a node in the graph, and δ is the degree of the deleted node.

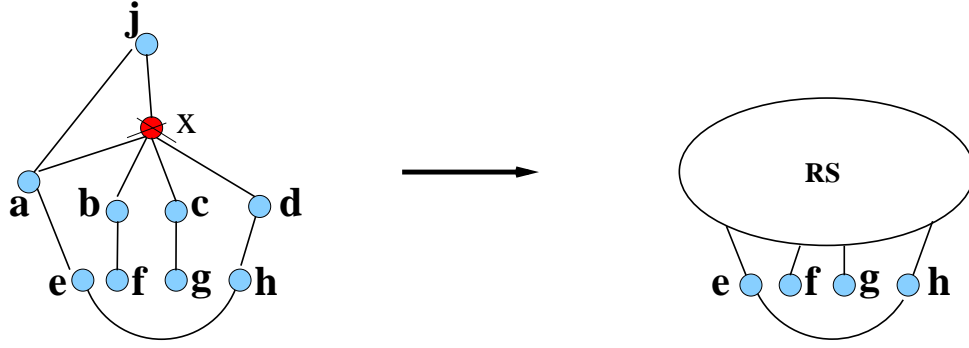


Figure 3: Deleted node x (in red, crossed) replaced by a Reconstruction Structure(RS), a structure formed by its neighbors (a, b, c, d, j) .

2 Virtual vs. real healing structures

Conceptually, Our algorithms use the same basic idea: when a node is deleted, replace it by a healing structure formed from its neighbors or nearby nodes, as shown in Figure 3. We can call this structure the ReconstructionStructure. Notice that we have defined Reconstruction Structures like a template and the exact structure is determined according to the desired properties of the algorithm. In most of our algorithms (DASH, ForgivingTree, Forgiving-Graph) the healing structure is a tree. In DASH, a balanced binary tree of neighbors with nodes arranged by previous degree increases is used. In the ForgivingTree [9](Section 3), another kind of binary balanced tree is used. In the Forgiving Graph [8](Section 4), our reconstruction tree is a *haft*(or half-full tree) (Section 4.1). In Xheal, the structure used is an expander.

It turns out that in these, trees are a natural choice for the graph properties we have tried to maintain. A balanced tree is a structure which has low distance between nodes (at most $2\log_2 n$ for a balanced binary tree) while each node has a small degree (at most 3 for a binary tree). At the same time, coming up with the suitable RTs and maintaining them over the run of the algorithm is quite a significant challenge. For Xheal, however, trees are not the right structure since the main property we are trying to heal here is edge expansion and such spectral properties, and trees do not have good edge expansion.

An idea that we have sometimes found very useful is the idea of using *virtual nodes*. A virtual node can be thought of as a marker in a reconstruction structure for a previously existing node in the network. A virtual node will be *simulated* by a real node (we call the existing non virtual nodes as real nodes). Informally, simulating would simply mean that the simulating node takes responsibilities of the connections attributed to the virtual node (more formally discussed later) In our algorithms, a virtual node is simulated by exactly one real node, but it may be possible to imagine algorithms where one virtual node may be simulated by multiple real nodes. Of course, one may have a single real node simulating multiple virtual nodes. In our algorithms, the resulting graph that we maintain is a mixture of real and virtual nodes. We call this a *virtual graph*. This is opposed to the *real graph*, which is the usual bijective mapping of the network to the graph with a processor mapping to a node and a connection mapping to an edge. Section 2.1 discusses the simple mapping

that gives the real graph from the virtual graph and also shows some simple properties helpful for bounding certain properties.

More formally, consider the actual graph $G(V, E)$ corresponding to the network, and a virtual graph $G'(V', E')$ with processors V' and edges E' . Consider a partition of the set V' into two sets V'_1 and V'_2 (possibly empty), with a surjective mapping $f : V'_2 \rightarrow V$ and a mapping $g : V'_1 \rightarrow V'_2$. The edge sets E' and E are related by the homomorphism given in Section 2.1. Then, we have:

real node: A node $v' \in V'_2$. By definition, we have a node $v \in V$, such that $f(v') = v$.

real edge: An edge $(v', w') \in E'$ such that both v' and w' are real nodes.

virtual node: Node $w' \in V'_1$. By definition, we have a node $v' = g(w')$, where $v' \in V'_2$.

We say the real node v' *simulates* virtual node w' .

virtual edge: An edge $(v', w') \in E'$ such that not both v' and w' are real nodes.

Virtual graphs are useful for a few conceptual reasons, some of which are:

- Virtual graphs may be easier to analyze and are good accountability structures (e.g. for bounding node degrees or distances). For example, if our Reconstruction Structure is a tree, and the virtual node is one of the internal nodes, we can claim that the node simulating it has increased its degree by at most 3 due to that virtual node.
- Virtual graphs may be easier to visualize. Sometimes, the real graph and its connections may look messy and the underlying pattern, if any, may be obscured. A well designed virtual graph scheme may give a clear insight into the structure and workings of the algorithm. For example, the ForgivingTree data structure is a virtual graph that is a tree (and the algorithm is a tree maintenance algorithm) whereas the real graph corresponding to the ForgivingTree may not be a tree at all.

Two of our algorithms, the Forgiving Tree and Forgiving Graph have reconstruction structures which use virtual nodes, and these structures are also trees. We call these reconstruction structures Reconstruction Trees and define them as follows:

Reconstruction Tree: A tree like structure (Figure 6) added by the healing algorithm on adversarial deletion of a single node and its edges. The reconstruction tree uses existing nodes (we call them *real nodes*) from the network and also may have virtual nodes i.e. nodes which are simulated by the real nodes in that reconstruction tree.

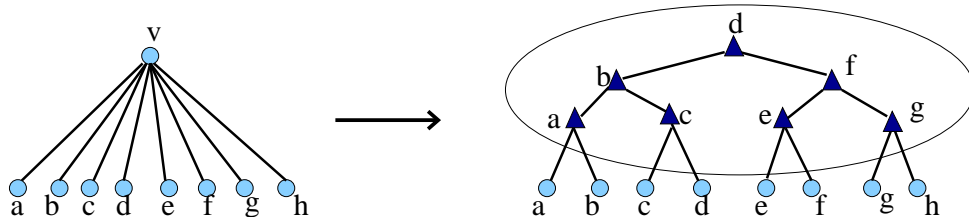


Figure 4: Deleted node v replaced by its Reconstruction Tree. The triangle shaped nodes are 'virtual' helper nodes simulated by the 'real' nodes which are in the leaf layer.

2.1 Real Graph from a virtual Graph

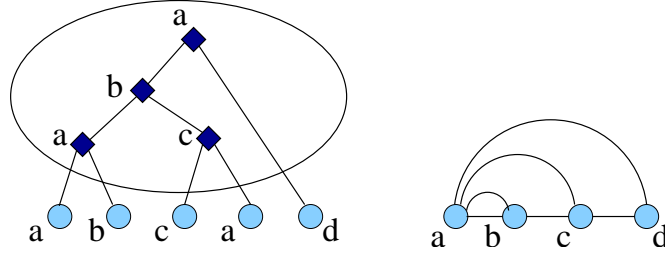


Figure 5: The actual graph (on the right) is a homomorphic image of a virtual graph (left) where the helper nodes are mapped to the nodes simulating them. Note both the node degrees and distances between nodes in the real graph cannot be more than those in the virtual Graph.

It is easy to see that a virtual graph maps to a real graph in a straightforward way: map all the virtual nodes to the real nodes simulating them. Figure 5 shows an example. More formally, the real graph is a homomorphic image of the virtual graph. Consider two graphs $G_1 = (V_1, E_1)$, and $G_2 = (V_2, E_2)$. In this context, a homomorphism may be defined as follows: A homomorphism is a function $f : V_1 \rightarrow V_2$ such that if undirected edge $\{v, w\}$ is in E_1 (the edge set of G_1) this implies that the edge $\{f(v), f(w)\}$ is in E_2 . Moreover, we say that G_2 is the homomorphic image of G_1 under f if the edges of G_2 are exactly the images of the edges of G_1 under the homomorphism. There can be multiple real and virtual nodes corresponding to a processor in the network that perform all the functions required of those nodes. Each node can be identified by its processor and some additional information. For node v in G_1 , let $Processor(v)$ be the name of that processor. In the real graph G_2 , there is only one node per processor and consider this node to be labelled with the name of that processor. Then, our homomorphism $H : V(G_1) \rightarrow V(G_2)$ is simply $H(v) = Processor(v)$.

Observation 1. For any graph homomorphism $F : G_1 \rightarrow G_2$, for all nodes u, v in V , $dist_{G_2}(F(u), F(v)) \leq dist_{G_1}(u, v)$ where $dist_G(x, y)$ is the distance between two nodes x and y in a graph G .

Observation 2. If the graph G_2 is the homomorphic image of graph G_1 under a graph homomorphism $F : G_1 \rightarrow G_2$, then for all nodes v' in G_2 , $deg_{G_2}(v') \leq \sum_{v \in F^{-1}(v')} deg_{G_1}(v)$, where $deg_G(x)$ is the degree of the node x in a graph G .

3 Forgiven Tree

The Forgiven Tree algorithm is in the same model as described in Section ?? except for the important difference that it does not handle node insertions. Also, the properties it heals are the diameter, degree and connectivity of the network. However, it is a very efficient algorithm for handling deletions and illustrative of the use of virtual graphs. The main procedure of Forgiven Tree is given as pseudocode as Algorithm 3.1. We only give the top procedure here and refer the reader to the paper [9] for the detailed pseudocode. At a high level, the Forgiven Tree works as follows. We begin with a rooted spanning tree T , which without loss of generality may as well be the entire network (Algorithm 3.1: Procedure INIT(T)). Each time a non-leaf node v is deleted, we think of it as being replaced


```

1: Given a tree  $T(V, E)$ 
2: INIT( $T$ ).
3: while true do
4:   if a vertex  $x$  is deleted then
5:     if children( $x$ ) is EMPTY then
6:       FIXLEAFDELETION( $x$ )
7:     else
8:       FIXNODEDELETION( $x$ )

```

Algorithm 3.1: FORGIVING TREE: The main function.

by a Reconstruction Tree (Section 2). For the Forgiving Tree, the Reconstruction Trees are actually balanced binary trees (with possibly an additional node as root). So, we define the following:

RT: In the ForgivingTree, let the reconstruction trees be denoted by RT. A RT is a balanced binary tree like structure (a balanced binary tree with a possible additional node as root) built after the deletion of a node (and its incident edges). The leaves of the tree are real nodes and all the internal nodes are virtual nodes simulated by those leaf nodes.

RT(v): RT(v) is the RT built on deletion of a node v . RT(v) is composed of the neighbors of v .

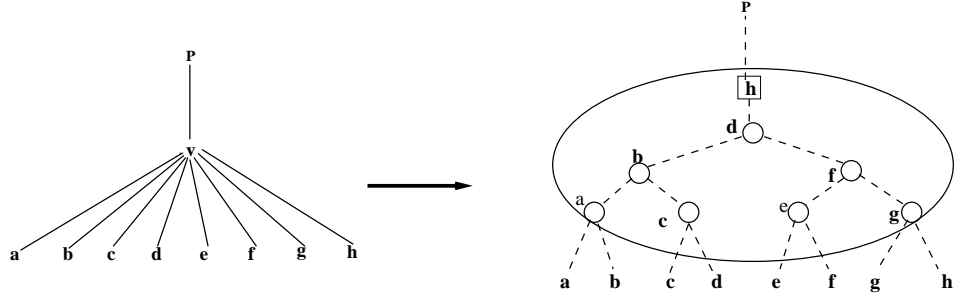


Figure 6: Deleted node v replaced by its Reconstruction Tree. The nodes in the oval are virtual nodes. Regular virtual nodes are depicted by circles and the heir virtual node by a rectangle.

Thus, on deletion of node v , we can imagine it being replaced by RT(v) (Implemented by FIXNODEDELETION(x): called from Algorithm 3.1). Depending on certain conditions explained later, the root of RT(v) could be a special node known as v 's *heir* (this will be discussed later). The root takes v 's place as the child of v 's parent. This is illustrated in figure 6. Note that each of the virtual nodes which was added is of degree 3, except the heir, if present.

When a leaf node is deleted, we do not replace it (Implemented by FIXLEAFDELETION(x): called from Algorithm 3.1). However, if the parent of the deleted leaf node was a virtual node, its degree has now reduced from 3 to 2, at which point we consider it redundant and “short-circuit” it, removing it from the graph, and connecting its surviving child directly to its parent. This helps to ensure that, except for heirs, every virtual node

is of degree exactly 3. One possibility is shown in Figure 7. There are a few cases of leaf deletion. We refer the reader to the paper for a full discussion.

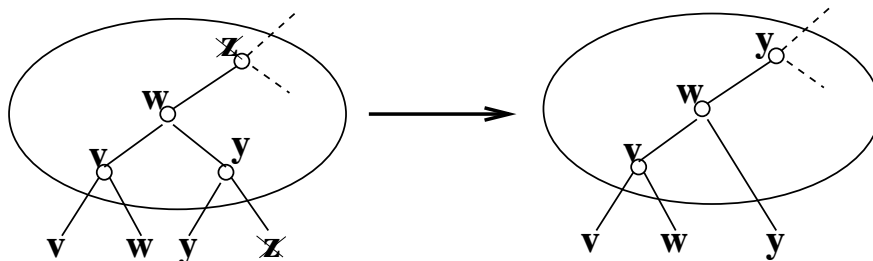


Figure 7: Leaf deletion: Parent of the deleted leaf node is a virtual node, and is ‘short circuited’.

After a long sequence of such deletions, we are left with a tree which is a patchwork mix of virtual nodes and original nodes. We note that the degrees of the original nodes never increase during the above procedure. Also, because the virtual trees are balanced binary trees, the deletion of a node v can, at worst, cause the distances between its neighbors to increase from 2 to $2\lceil \log d \rceil$, where d is the degree of v . This ensures that, even after an arbitrary sequence of deletions, the distance between any pair of surviving actual nodes has not increased by more than a $\lceil \log \Delta \rceil$ factor, where Δ is the maximum degree of the original tree.

Since our algorithm is only allowed to add edges and not nodes, we cannot really add these virtual nodes to the network. We get around this by assigning each virtual node to an actual node, and adding new edges between actual nodes in order to allow “simulation” of each virtual node (Section 2). More precisely, our actual graph is the homomorphic image of the tree described above, under a graph homomorphism which fixes the actual nodes in the tree and maps each virtual node to a distinct actual node which is “simulating” it (Section 2.1). Note that, because each actual node ever simulates at most one virtual node at a time, and virtual nodes have degree at most 3, this ensures that the maximum degree increase of our algorithm is at most 3.

The heart of our algorithm is a very efficient distributed algorithm for keeping track of which actual node is assigned to simulate each virtual node, so that the replacement of each deleted node by its virtual tree can be done in $O(1)$ time. We accomplish this using a system of “wills,” in which each vertex v instructs each of its children (or their “heirs”) in the event of v ’s deletion, how to simulate the virtual tree replacing v , and also the virtual node v was simulating (if any).

This will is prepared in advance, before v ’s deletion, and entrusted to v ’s children or their surviving heirs. An example of this is shown in figure 8. Certain events, such as the deletion of one of v ’s children, or a change in which virtual node v is simulating, may cause v to revise its will, informing the affected children or their surviving heirs. As proven in [9], the total number of messages and node IDs which must be sent is $O(1)$ per deleted vertex; the number of bits sent is thus $O(\log n)$. In addition, there is a startup cost for communicating the initial wills: this is $O(1)$ latency; and $O(1)$ messages and $O(\log n)$ bits per edge in the original network.

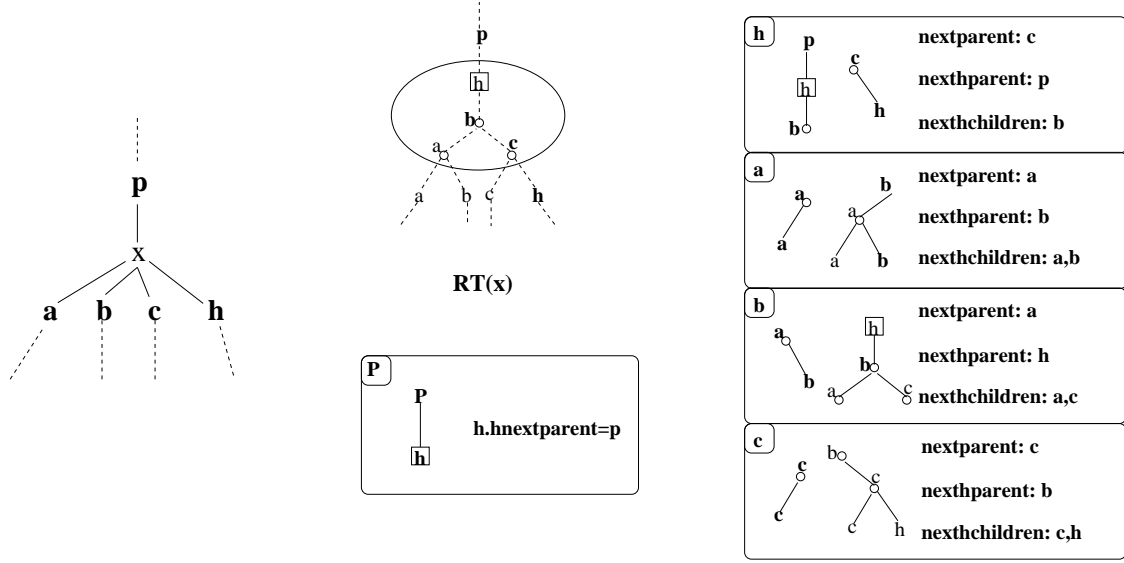


Figure 8: The leftmost column shows a small segment of the network. The $RT(x)$ corresponding to this figure is shown. Every neighbor of node x stores the portion of $RT(x)$ relevant to it. Each rectangular box is labelled with a neighbor and shows the portions and the value of the corresponding fields .

3.1 Forgiving Tree success metrics

Here, we state a formal theorem regarding the properties of the Forgiving Tree (For the proof, we refer the reader to [9] or [17]). The result shows that Forgiving Tree can self-heal the diameter of the graph very efficiently with only a constant additive increase in node degrees over all sequence of deletions. We also state a lower bound (with proof) that is, in fact, matched by our algorithm.

Theorem 1. *The Forgiving Tree has the following properties:*

1. *The Forgiving Tree increases the degree of any vertex by at most 3.*
2. *The Forgiving Tree always has diameter $O(D \log \Delta)$.*
3. *The latency per deletion and number of messages sent per node per deletion is $O(1)$; each message contains $O(1)$ node IDs and thus $O(\log n)$ bits.*

Theorem 2. *Consider any self-healing algorithm that ensures that: 1) each node increases its degree by at most α , for some $\alpha \geq 3$; and 2) the diameter of the graph increases by a multiplicative factor of at most β . Then for any positive Δ , for some initial graph with maximum degree Δ , it must be the case that $\beta \geq \frac{1}{2}[\log_{\alpha+1} \Delta - 1]$.*

Proof. Let G be a star on $\Delta + 1$ vertices, where x is the root node, and x has Δ edges with each of the other nodes in the graph. Let G' be the graph created after the adversary deletes the node x . Consider a breadth first search tree, T , rooted at some arbitrary node y in G' . We know that the self-healing algorithm can increase the degree of each node by at most α , thus the root node in T can have at most $\alpha + 1$ children, and other nodes can have at most

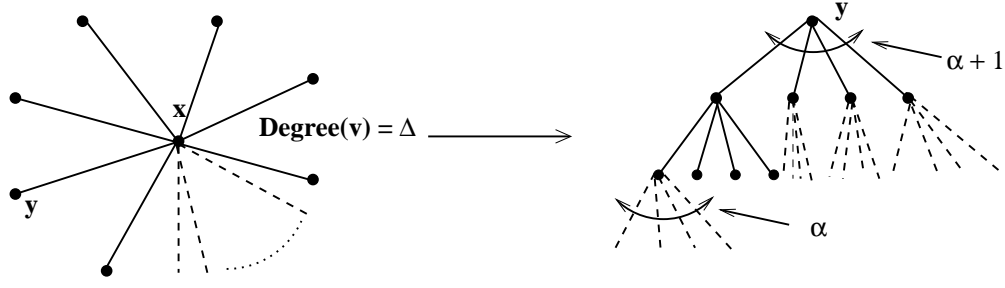


Figure 9: Deletion of the central node v of a star leads to an increase in the diameter. Here, the healing algorithm increases the degree of any node by at most α .

α children. Let h be the height of T . Then we know that $1 + (\alpha + 1) \sum_{i=0}^{h-1} \alpha^i \geq \Delta$. This implies that $(\alpha + 1)^{h+1} \geq \Delta$ for $\alpha \geq 3$, or $h + 1 \geq \log_{\alpha+1} \Delta$. Since the diameter of G is 2, we know that $\beta \geq h/2$, and thus $2\beta + 1 \geq \log_{\alpha+1} \Delta$. Rearranging, we get $\beta \geq \frac{1}{2}[\log_{\alpha+1} \Delta - 1]$. This is illustrated in figure 9. \square

We note that this lower-bound compares favorable with the general result achieved with our data structure. The Forgiven Tree can be modified so that it ensures that 1) the degree of any node increases by no more than α for any $\alpha \geq 3$; and that the diameter increases by no more than a multiplicative factor of $\beta \leq 2 \log_{\alpha} \Delta + 2$.

4 Forgiven Graph

This section gives an overview of the *Forgiven Graph* [8]. The Forgiven Graph was a successor of the Forgiven Tree [9]. The improvements of the Forgiven Graph over the Forgiven Tree are threefold. First, the Forgiven Graph maintains low stretch i.e. it ensures that the distance between any pair of nodes v and w is close to what their distance would be even if there were no node deletions. It ensures this property even while keeping the degree increase of all nodes no more than a multiplicative factor of 3. Moreover, this tradeoff between stretch and degree increase is asymptotically optimal. Second, the Forgiven Graph handles both adversarial insertions and deletions, while the Forgiven Tree could only handle adversarial deletions (and no type of insertion). Finally, the Forgiven Graph requires only a very simple and minimal initialization phase, while the Forgiven Tree required an initialization phase which involved sending $O(n \log n)$ messages, where n was the number of nodes initially in the network, and had a latency equal to the initial diameter of the network. Additionally, the Forgiven Graph is divergent technically from the Forgiven Tree, it makes significant use of a novel distributed data structure that we call a Half-full Tree or “haft”. hafts are discussed in Section 4.1. Hafts are mergable trees and their properties allow us to construct the Forgiven Graph algorithm as an efficient, distributed data structure with the desired properties.

Here, we give a high level description of our algorithm. An adversary can effect the network in one of two ways: inserting a new node in the network or deleting an existing node from the network. Node insertion is straightforward and is dependent on the specific policies of the network. When an insertion happens, our incoming node and its neighbors update the data structures that are used by our algorithm.

Each time a node v is deleted, we can think of it as being replaced by a Reconstruction Tree(Section 2) . For this algorithm, let us call the particular types Reconstruction trees used as RTs. The RT is a haft(Section 4.1) made up using virtual nodes (if needed). Formally:

RT: In the ForgivingGraph, RT is a haft built after the deletion of a node (and its incident edges). The leaves of the haft are real nodes and all the internal nodes are virtual nodes simulated by those leaf nodes.

RT(v): RT(v) is the RT built on deletion of a node v . RT(v) is composed of the neighbors of v and of nodes from RTs in which v was a member.

A single real node itself is a trivial RT with one node. RT(v) is formed by merging all the neighboring RTs of v using the strip and merge operations from Section 4.2. Thus, following a deletion, we may have a graph with both real and virtual nodes. After a long sequence of such insertions and deletions, this graph is a patchwork mix of virtual nodes and real nodes. Let us call this graph FG (short for ForgivingGraph). As for the other graphs, FG_T is the graph FG at time T .

It is easy to show that each virtual node has a degree of at most 3 (since they are internal nodes of binary trees). In our algorithm, we allow a real node to simulate at most one virtual node per deleted neighbor, thus bounding its degree increase. Also, because the hafts are balanced binary trees, the deletion of a node v can, at worst, cause the distances between its neighbors to increase from 2 to $2\lceil \log d \rceil$ by traveling through RT(v), where d is the degree of v in G' (the graph consisting solely of the original nodes and insertions without regard to deletions and healings). However, since this deletion may cause many RTs to merge and the new RT formed may involve all the nodes in the graph, the distances between any pair of actual surviving nodes may increase by no more than a $\lceil \log n \rceil$ factor.

4.1 Half-Full Trees

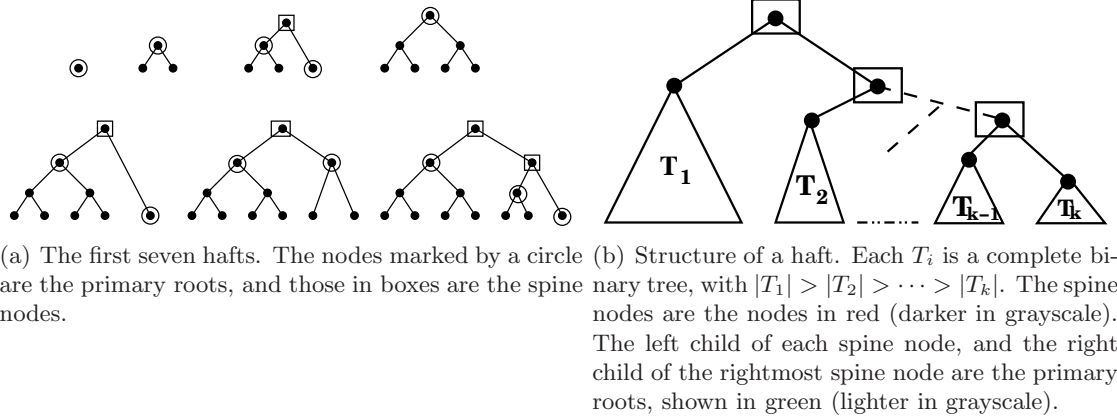


Figure 10: haft (half-full tree)

In this section, we define half-full trees (or hafts, for short), and describe their most important properties for our present application. These trees are similar to the “staircase trees” described by Vaucher [18]. However, our presentation will be self-contained. Hafts

also evoke the flavor of other well known data structures such as binomial trees and binomial heaps. A comparison is given at the end of this section.

Half-full tree: A *half-full tree*, or *haft*, is a rooted binary tree in which every non-leaf node v has the following properties:

- v has exactly two children.
- The left child of v is the root of a complete binary subtree that contains at least half of v 's descendants.

Primary root: A *primary root* is a node in a haft such that:

- It is the root of a complete subtree.
- Its parent, if it has one, is not the root of a complete subtree.

Spine: A *spine node* is the parent of a primary root. Equivalently, it is a node in a haft which is not the root of a complete subtree. The *spine* of a haft is the set of all spine nodes. We observe that the spine, if non-empty, consists of the vertices of a path, with the root of the haft as one endpoint.

Figure 10(a) shows several examples of hafts. We now give a simple structural lemma which completely characterizes any haft as a function of the number of its leaves. This will be useful later when we wish to perform merging operations on the hafts used by our algorithm.

Lemma 1 (Binary representation of Hafts). *Let ℓ be a positive integer. Then there is a unique haft T having ℓ leaves. Moreover, let h be the number of ones in the binary representation of ℓ , and suppose $x_1 > x_2 > \dots > x_h$ are the indices of these ones, so that*

$$\ell = \sum 2^{x_i}.$$

Then either

- $h = 1$, and T is a complete tree of depth x_1 , or
- $h \geq 2$, and T consists of $h - 1$ spine nodes s_1, \dots, s_{h-1} , together with h complete binary trees T_1, \dots, T_h , where
 - s_1 is the root of T ,
 - each T_i has depth x_i ,
 - each s_i has the root of T_i as its left child
 - for $1 \leq i \leq h - 2$, s_i has s_{i+1} as its right child
 - s_{h-1} has the root of T_h as its right child

Corollary 1. *Let T be a haft having ℓ leaves. Then the depth of T equals $\lceil \log \ell \rceil$.*

Proof of Lemma 1. We will prove the detailed structure of T , from which the uniqueness follows directly.

First, consider the case $h = 1$ (i.e., ℓ is a power of 2). If $\ell = 1$, there is nothing to prove. Assume $\ell > 1$. Now the left subtree of T is complete, and hence has number of leaves equal to a power of two. Since at least half of the leaves are on the left subtree, this power of two is at least $\ell/2$. Since the root of T has two children, not all of the leaves are on the left

subtree, and hence there are exactly $\ell/2$ leaves on the left subtree, and thus also $\ell/2$ leaves on the right subtree. Since it is immediate from the definition that any subtree of a haft is also a haft, it follows by induction on ℓ (being a power of two) that the right subtree is also a complete subtree. Thus, T is complete.

Now, suppose $h \geq 2$. Let us denote the root of T by s_1 . Because ℓ is not a power of 2, s_1 must be a spine node. Since the left subtree, T_1 , is complete and contains between $\ell/2$ and ℓ leaves, it must have depth x_1 . Since the right subtree is a haft having number of leaves equal to

$$\ell - 2^{x_1} = \sum_{i=2}^h 2^{x_i}$$

it follows by induction on ℓ (being any positive integer) that it has the claimed structure. Thus, T is also as claimed. \square

4.2 Operations on Hafts

We define the following operations on hafts:

1. *Strip*: Suppose T is a haft with h ones in its binary representation. The Strip operation removes $h - 1$ nodes from T returning a forest of h complete trees.
2. *Merge*: The Merge operation joins hafts together using additional isolated single nodes, to create a single new haft.

We now describe these operations in more detail:

4.2.1 STRIP

By Lemma 1, if we remove the spine from a haft, T , we are left with a forest of h complete binary trees, where h is the number of ones in the binary representation of the number of leaves of T . The operation $\text{Strip}(T)$ returns this forest.

The Strip operation works as follows: If T is a complete tree, then return T itself. Note that the root of the T is the only primary root in this case. If T is not a complete tree, then F is obtained as follows. Starting from the root of T , traverse the direct path towards the rightmost leaf of T . Remove a node if it is not a primary root. Stop when a primary root or a leaf node (which is a primary root too) is discovered. In figure 10(b) the Strip operation removes the nodes indicated by the square boxes. Figure 11 shows the use of the Strip operation.

We now prove why the Strip operation works.

Lemma 2. *The Strip operation returns the subtrees rooted at all primary roots in the input haft.*

Proof. By the definitions of haft and primary root, if a vertex is not the root of a complete subtree, its left child is guaranteed to be a primary root. Thus, either the root of the haft is a primary root or its left child is. If the left child is a primary root, there can be no other primary root in the left subtree, so we return the tree rooted at that child. Recursively applying the same test to the right child, we get all the primary roots. \square

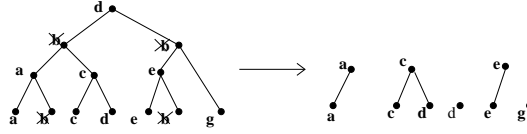


Figure 11: Deletion of a node and its virtual nodes lead to breakup of RT into components. The Strip operation or a simple variant (for non-hafts) returns a set of complete trees, which can then be merged.

4.2.2 MERGE

By Lemma 1, every haft is completely characterized by its number of leaves. Merging hafts is analogous to binary addition of these numbers. The new binary number obtained is the number of leaves in the haft produced by the Merge operation. This is illustrated in figure 12.

The first step of the Merge operation is to apply the Strip operation on the input trees. This gives a forest of complete trees. These complete trees can be recombined with the help of extra nodes to obtain a new haft. Let $Size(X)$ be the number of nodes in a tree X . Consider two complete trees T_1 and T_2 ($Size(T_1) > Size(T_2)$), with roots r_1 and r_2 respectively, and an extra node v . To merge these trees, make r_1 the left child and r_2 the right child of v by adding edges between them. The merged tree is always a haft. Thus, the merge operation $Merge(haft_1, haft_2, \dots)$ is as follows:

1. Apply Strip to all the hafts to get a forest of complete trees.
2. Let T_1, T_2, \dots, T_k be the k complete trees sorted in ascending order of their size. Traverse the list in ascending order; let T_i and T_{i+1} be the first two adjacent trees of the same size and v be a single isolated vertex, join T_i and T_{i+1} by making v the parent of the root of T_i and the root of T_{i+1} , to give a new tree. Reinsert this tree in the correct place in the sorted list. Continue traversal of the list from the position of the last merge, joining pairs of trees of equal sizes. At the end of this traversal, we are left with a sorted list of complete trees, all of different sizes.
3. Let T_1, T_2, \dots, T_l be the sorted list of complete trees obtained after the previous step. Traverse the list in ascending order, joining adjacent trees using single isolated vertices. Let w be a single isolated vertex. Join T_1 and T_2 by making the root of T_2 the left child and the root of T_1 the right child of w , respectively. This gives a new haft. Join this haft and T_3 by using another available isolated vertex, making the larger tree (T_3) its left child. Continue this process till there is a single haft.

4.3 hafts vs binomial heaps

Half-full trees are similar to binomial heaps [3] in the sense that both are mergable structures and their representation and merge have correspondence to binary numbers and binary addition. However, there are clear differences. Binomial heaps satisfy the heap property, which hafts (at least in our application) need not. During merging, two binomial heaps or trees are joined directly by connecting their roots whereas in the merge for hafts, an additional

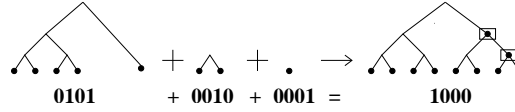


Figure 12: Merging three hafts. The vertices in the square boxes are the new isolated vertices used to join the complete. The square shaped vertices are the isolated vertices used to join the complete trees. Merging is analogous to binary number addition, where the number of leaves are represented as binary numbers.

node is needed as a new root for the merging of two hafts. Also, binomial trees/heaps are not binary trees whereas hafts, by definition, are binary trees. For comparison purposes, let us call the number of leaves of a haft as its order since the shape of each haft is determined by its number of leaves. Then, a haft with order k has $2k - 1$ nodes whereas a binomial tree of order k has 2^k nodes. A binomial tree has a root node whose children are roots of binomial trees of smaller orders sorted by their order, whereas in a haft the spine (as defined before) has, as children, complete trees (which are a special case of hafts) of smaller orders also arranged in a sorted order.

4.4 FG: Distributed implementation

In this section, we give a brief overview of how the Forging Graph can be implemented in a completely distributed fashion. Due to space constraints, we refer the reader to the paper for the complete details.

As mentioned earlier, deletion of a node v leads to it being replaced by a Reconstruction Tree (RT or $RT(v)$, for short) in G . Recall that the graph G' is the graph consisting of solely the original nodes and insertions (Figure 1). Figure 13 shows a small series of deletions and repairs by the ForgingGraph algorithm. Notice that after healing on the third deletion some nodes are occurring as leaf nodes multiple times (Figure 13(f)). Here, edge information is useful for differentiating between these nodes. A real node takes part in a nontrivial RT (i.e. a RT with more than one node) only if one of its neighbors got deleted. It can have k edges into RTs only if k of its neighbors have already been deleted, for some integer k . Each edge from a real node into a RT corresponds to a deleted neighbor. We can imagine this edge never got deleted and just that its other endpoint got replaced by a virtual node. Thus, if there was an edge between nodes x and y , and node y got deleted, we can keep this edge labelled as (x, y) . Alternatively, the edge is labelled with it's name in G' , which will always be (x, y) since G' has no deletions. For convenience, when a node occurs as a leaf node multiple times in a RT, we will often consider each occurrence as a separate node and describe it as such. Figure 14 shows this alternate representation. Notice that it is easy to see the haft structure in this representation and we stay in the realm of trees. From now on, when we refer to a leaf node of a RT, we will mean a real node augmented with the edge information. Thus, when we state that there is at most one virtual node corresponding to a leaf node of a RT, this is equivalent to saying that there is at most one virtual node in a RT corresponding to an edge in the graph G' . The actual processor on which we are executing the algorithm must keep track of its real nodes, edges and virtual nodes.

Insertion is straightforward; a node simply comes in with its edges and the effected nodes update their relevant information. At a high level, when a node is deleted, the algorithm

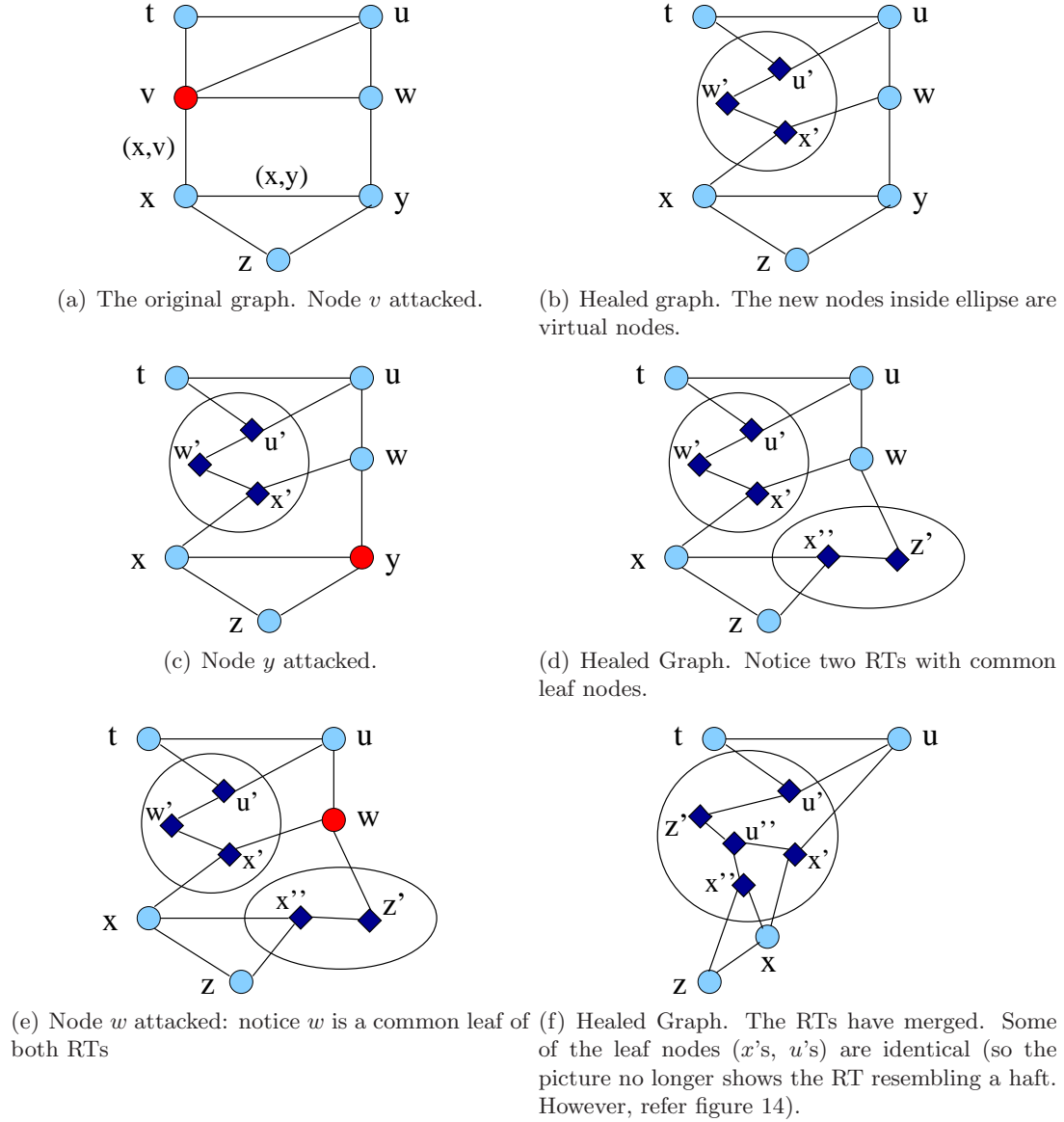


Figure 13: Effect of 3 deletions on a graph. The RT for each deleted node consists of the virtual nodes, plus the neighbors of the deleted node which form the leaves of the tree. In this example, the deleted nodes form an independent set, so the structure of the RTs does not depend on the deletion order.

for repair is as shown in Algorithm 4.1. The repair proceeds in two parts. The first part is a quick $O(1)$ phase in which the neighbors of the deleted node connect themselves in the form of a binary tree. Consider the effect of the deletion of v on one of the RTs of which v is a leaf. Removal of this leaf and of the virtual node corresponding to that leaf (if any) splits this RT into connected components. We select particular nodes which were neighbors of the deleted nodes from each of these components. Let $Nset$ be the collection of all these nodes together

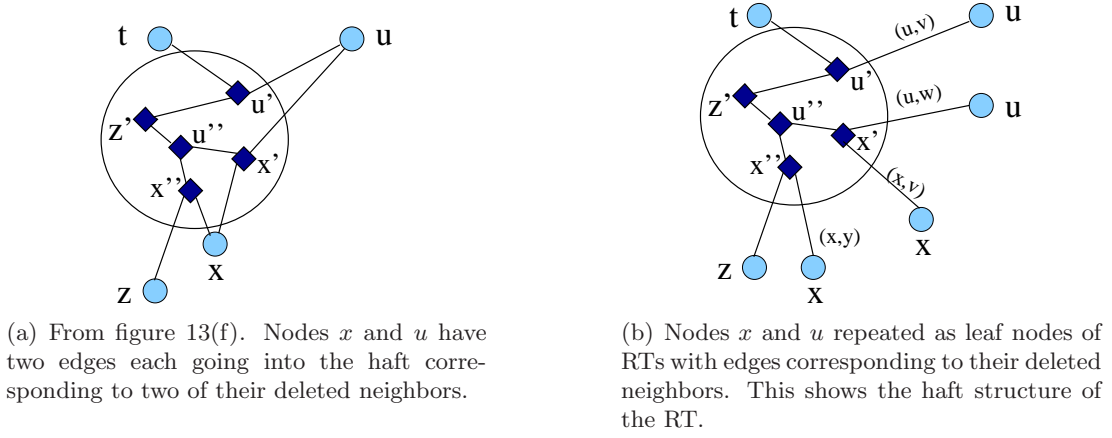


Figure 14: Equivalent Representations of a RT.

with any undeleted neighbors of v in FG. We shall call a component taking part in the merge process (irrespective of whether it is a haft or not) as a *RTfragment*, to distinguish it from the final RT formed at the end of the merge process. In part 2 of the repair, the RTfragments are merged (Figure 15). Before we can reconnect these RTfragments into a single haft, we need to further break them up into hafts (more specifically, we actually break them into complete trees which are also a kind of haft) so that we can merge them.

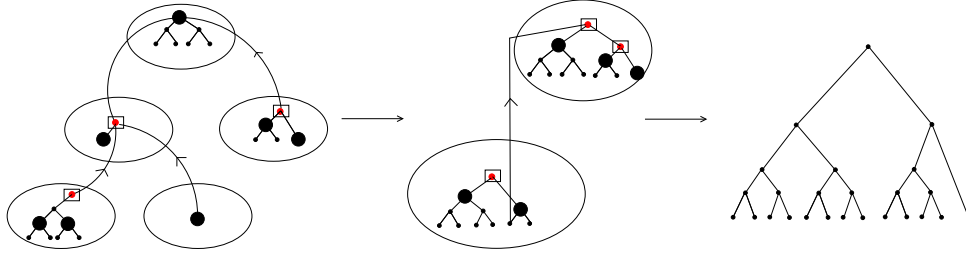


Figure 15: On deletion of a node v , The RTfragments to be merged are connected by a binary tree BT_v . The leaf RTfragments merge with their parents till a single RT is left. The solid circles are the primary roots. The (red color) nodes in the square boxes are spine nodes removed at each step.

4.5 ForgingGraph success metrics

We state here a theorem summarizing the properties of Forging Graph (for the proof, refer to the paper). We also state a lower bound somewhat similar to theorem 2.

Theorem 3. *The Algorithm ForgingGraph has the following properties:*

1. Degree increase: *For any node v in $V(G_T)$, after any number of time steps, T , the degree of v in G_T is at most 3 times the degree of v in G'_T .*
2. Stretch: *For any nodes x, y in $V(G_T)$, after any number of time steps, T , the distance between x and y in G_T is at most $\log(n)$ times the distance in G'_T .*

Part 1: *Deletion of node v splits RTs into RTfragments*

- 1: Anchors(designated nodes) from each RTfragment combine to form a binary tree BT_v

Part 2:

- 1: Each anchor will initiate 2-phase primary roots discovery
 - i) Identify primary roots in their RTfragments (may have false positives)
 - ii) Exchange primary roots lists and broadcast correct list through BT_v
- 2: Merge leaves of BT_v with parent node, in parallel, to get $BT_{v'}$ (anchors may change)
- 3: Anchors do root discovery (no false positives now), and merge leaves with parent
- 4: Repeat previous step till a single RT left

Algorithm 4.1: ForgivingGraph actions after node deletion (high level pseudocode)

3. Cost: *After each deletion, the repair phase requires the sending of at most $O(d \log n)$ messages, each of length $O(d \log n + \log^2 n)$. Moreover, this can be done in parallel by the neighbors of the deleted node, in time $\text{polylog}(d, n)$.*

Theorem 4. *Let n be a positive integer, $\alpha \geq 3$ and $\beta = \frac{1}{2}(\log_\alpha(n-1) - 1)$. Then there exists a graph on n vertices and a vertex deletion such that any way of repairing this deletion under our model must either increase the degree of some node by more than a factor of α , or it must increase the distance between some pair of nodes by at least a factor of β .*

5 Conclusion and notes

This chapter discussed self-healing in dynamic networks and introduced a responsive and scalable approach towards self-healing in reconfigurable networks. A model of self-healing and recent algorithms on self-healing using this or similar models were succinctly compared. A generic idea of using virtual structures (which may be useful for many problems besides self-healing) was introduced. Two algorithms: Forgiving Tree [9] and ForgivingGraph [8] were discussed in more detail, with particular emphasis on their use of virtual graphs. The readers are encouraged to refer to an earlier algorithm in this line of work: DASH [14], and the latest algorithm: Xheal [12], both of which do not use virtual structures.

References

- [1] Villu Arak. What happened on August 16, August 2007. <http://heartbeat.skype.com/2007/08/what-happened-on-august-16.html>.
- [2] Andrew Berns and Sukumar Ghosh. Dissecting self-* properties. *Self-Adaptive and Self-Organizing Systems, International Conference on*, 0:10–19, 2009.
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, second edition, 2001.
- [4] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, November 1974.
- [5] Shlomi Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.

- [6] Shlomi Dolev and Nir Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. *Theor. Comput. Sci.*, 410(6-7):514–532, 2009.
- [7] Ken Fisher. Skype talks of "perfect storm" that caused outage, clarifies blame, August 2007. <http://arstechnica.com/news.ars/post/20070821-skype-talks-of-perfect-storm.html>.
- [8] Thomas P. Hayes, Jared Saia, and Amitabh Trehan. The forgiving graph: a distributed data structure for low stretch under adversarial attack. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 121–130, New York, NY, USA, 2009. ACM.
- [9] Tom Hayes, Navin Rustagi, Jared Saia, and Amitabh Trehan. The forgiving tree: a self-healing distributed data structure. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 203–212, New York, NY, USA, 2008. ACM.
- [10] Om Malik. Does Skype Outage Expose P2Ps Limitations?, August 2007. <http://gigaom.com/2007/08/16/skype-outage>.
- [11] Matt Moore. Skype's outage not a hang-up for user base, August 2007. <http://www.usatoday.com/tech/wireless/phones/2007-08-24-skype-outage-effects-N.htm>.
- [12] Gopal Pandurangan and Amitabh Trehan. Xheal: localized self-healing using expanders. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11, pages 301–310, New York, NY, USA, 2011. ACM.
- [13] Bill Ray. Skype hangs up on users, August 2007. http://www.theregister.co.uk/2007/08/16/skype_down/.
- [14] Jared Saia and Amitabh Trehan. Picking up the pieces: Self-healing in reconfigurable networks. In *IPDPS. 22nd IEEE International Symposium on Parallel and Distributed Processing.*, pages 1–12. IEEE, April 2008.
- [15] Brad Stone. Skype: Microsoft Update Took Us Down, August 2007. <http://bits.blogs.nytimes.com/2007/08/20/skype-microsoft-update-took-us-down>.
- [16] Gerard Tel. *Introduction to distributed algorithms*. Cambridge University Press, New York, NY, USA, 1994.
- [17] Amitabh Trehan. *Algorithms for self-healing networks*. Dissertation, University of New Mexico, 2010.
- [18] Jean G. Vaucher. Building optimal binary search trees from sorted values in $O(n)$ time. In *Essays in Memory of Ole-Johan Dahl*, pages 376–388, 2004.